Nolan Stevenson, 10/8/2022
ASEN 6055, HW3

Problem 1

Use: $x^b = \begin{bmatrix} 1 \\ -1 \\ 2.5 \end{bmatrix}$ $B = \begin{bmatrix} 0.1 & 0.02 & 0 \\ 0.02 & 0.1 & 0.01 \\ 0 & 0.01 & 0.2 \end{bmatrix}$ $y = 2$ w/ $R = [0.04^2]$

where $y = H(x) + \epsilon = e^{x_1 x_3} + \ln(1 + x_3^2) + \epsilon$

Minimize $J(x) = \frac{1}{2}(y - H(x))^T R^{-1}(y - H(x)) + \frac{1}{2}(x - x^b)^T B^{-1}(x - x^b)$

1.) Newtons Method

$x_0 = \begin{bmatrix} 0.5 \\ -0.5 \\ 1 \end{bmatrix}$ for $x_{k+1} = x_k - \frac{\nabla J(x)}{\nabla^2 J(x)} = x_k - (\nabla^2 J(x))^{-1}(\nabla J(x))$

Iteratively run for $n = 100$ max steps
until w/in 1e-6 tol of step

After 8 iterations, we converged to:

$x^* = \begin{bmatrix} 1.0685 \\ -1.0862 \\ 2.1032 \end{bmatrix}$ w/ a cost, J^*, of 0.4557

Quite small!

Now, try $x_0 = \begin{bmatrix} 2.5 \\ -2.5 \\ -1 \end{bmatrix}$

we get 17 iterations for $x^* = \begin{bmatrix} 1.0685 \\ -1.0862 \\ 2.1032 \end{bmatrix}$, and $J^* = 0.4557$

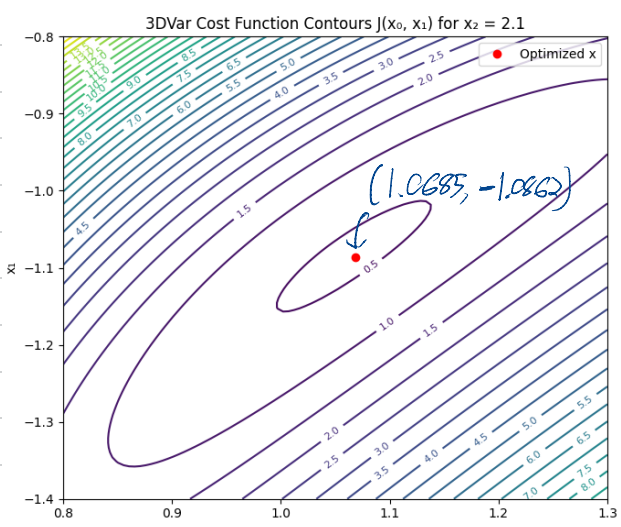
so exact same final converged value

And w/ $x_0 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

w/ also get

$x^* = \begin{bmatrix} 1.0685 \\ -1.0862 \\ 2.1032 \end{bmatrix}$, and $J^* = 0.4557$ after 10 iterations

We can visualize this by plotting the 2D slice @ $x_2 = 2.1$:



Looks like we chose a good minima!

Right on func min for a $x_2 = 2.1$ slice

2.) Gradient Descent

Where $x_{k+1} = x_k - \alpha \nabla J(x_k)$

Will use Quadratic Optimization to auto choose α.

Where $\alpha_k = \frac{g_k^T g_k}{g_k^T A g_k}$ for $\nabla J = Ax_k + b$.

↓

Thus, can get A from hessian, $\nabla^2 J = A$

So, $\alpha_k = \frac{g_k^T g_k}{g_k^T H_k g_k}$ where $g_k = \nabla J$ and $H_k = \nabla^2 J$

Thus, iteratively $x_{k+1} = x_k - \alpha_k \cdot g_k$

From $x_0 = \begin{bmatrix} 0.5 \\ -0.5 \\ 1 \end{bmatrix}$, it took **44 steps** to get to

$$x^* = \begin{bmatrix} 1.06845 \\ -1.08621 \\ 2.10315 \end{bmatrix} \text{ and } J^* = 0.45571$$

3.) Discuss

Thus, both Newtons and Quadradic GD both converged to a optimal solution of $x^* = [1.068, -1.086, 2.103]$ and $J^* = 0.4557$.

But, Newtons converged in only 8 iterations compared to 44 for GD. Usually, gradient descent is less computation than Newtons, as you don't need to calculate a Hessian. However, this function is complex enough to optimize that a fixed step wasn't possible.

Thus, need quadradic solution for α^*, using Hessian anyways.

The alpha used was:

So very hard to pick fixed step.

```
Newton converged in 8 iterations.
Alpha: [[0.00118144]]
Alpha: [[0.00191912]]
Alpha: [[0.0061853]]
Alpha: [[0.00172634]]
Alpha: [[0.00790795]]
Alpha: [[0.00158968]]
Alpha: [[0.01063382]]
Alpha: [[0.00150229]]
Alpha: [[0.01515953]]
Alpha: [[0.00146789]]
Alpha: [[0.02261922]]
Alpha: [[0.00150092]]
Alpha: [[0.03275687]]
Alpha: [[0.0016081]]
Alpha: [[0.04064522]]
Alpha: [[0.00174036]]
Alpha: [[0.04318666]]
Alpha: [[0.00183602]]
Alpha: [[0.04361837]]
Alpha: [[0.00189045]]
Alpha: [[0.0436608]]
Alpha: [[0.00191896]]
Alpha: [[0.04363933]]
Alpha: [[0.00193347]]
Alpha: [[0.04350485]]
Alpha: [[0.00194097]]
Alpha: [[0.04284176]]
Alpha: [[0.00194593]]
Alpha: [[0.04270474]]
Alpha: [[0.00194823]]
Alpha: [[0.04226357]]
Alpha: [[0.00195042]]
Alpha: [[0.0433417]]
Alpha: [[0.0019489]]
Alpha: [[0.04282187]]
Alpha: [[0.00195102]]
Alpha: [[0.06073334]]
Alpha: [[0.00192547]]
Alpha: [[0.03709946]]
Alpha: [[0.00196719]]
Alpha: [[0.06875346]]
Alpha: [[0.00192229]]
Alpha: [[0.00754191]]
Alpha: [[0.00216622]]
Steepest Descent converged in 44 iterations.
```

Thus, generally Newtons is faster at converging but does require Jacobian and Hessian.

Gradient Descent can be faster per iteration, but requires more steps, Jacobian and tuning of α. Or using 1-D line search for α does solve lots of GD issues, but requires lots of computation.

In this case, both did converge to optimal

Problem 2

Develop and test tangent linear and adjoint code.

- Adjoint for gradient of J if $\frac{\partial H}{\partial x}$ for available

Where $H = e^{x_1 x_2} + \ln(1 + x_3^2)$

and $\nabla H = \left[x_2 e^{x_1 x_2},\ x_1 e^{x_1 x_2},\ 2x_3 \cdot \frac{1}{1+x_3^2} \right]$

For tangent statement $\lim_{\delta x \to 0} \frac{H(x+\delta x) - H(x)}{H_{TLM}(x, \delta x)} = 1$

$$\begin{bmatrix} \delta x_1 \\ \delta x_2 \\ \delta x_3 \\ \delta y \end{bmatrix}_I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x_2 e^{x_1 x_2} & x_1 e^{x_1 x_2} & \frac{2x_3}{1+x_3^2} & 0 \end{bmatrix} \begin{bmatrix} \delta x_1 \\ \delta x_2 \\ \delta x_3 \\ \delta y \end{bmatrix}_{I-1}$$

Tangent

Using form from class

and adjoint form:

$$\begin{bmatrix} \delta x_1^* \\ \delta x_2^* \\ \delta x_3^* \\ \delta y^* \end{bmatrix}_{I-1} = \begin{bmatrix} 1 & 0 & 0 & x_2 e^{x_1 x_2} \\ 0 & 1 & 0 & x_1 e^{x_1 x_2} \\ 0 & 0 & 1 & \frac{2x_3}{1+x_3^2} \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \delta x_1^* \\ \delta x_2^* \\ \delta x_3^* \\ \delta y^* \end{bmatrix}_I$$

Adjoint

2.) Program forward, tangent linear, and adjoint

To test at $x_0 = [0\ 0\ 1]$ w/ $\delta x = [0.01, 0.01, 0.01]$

Forward: $y = H(x) = e^{x_1 x_2} + \ln(1 + x_3^2)$

and Jacobian, $\nabla H = \left[x_2 e^{x_1 x_2},\ x_1 e^{x_1 x_2},\ 2x_3 \cdot \frac{1}{1 + x_3^2} \right]$

Thus,

```python
# Forward model
def H(x):
    return np.exp(x[0] * x[1]) + np.log(1 + x[2]**2)

def H_jacobian(x):
    return np.array([x[1] * np.exp(x[0] * x[1]), x[0] * np.exp(x[0] * x[1]), 2 * x[2] / (1 + x[2]**2)])
```

Tangent Linear want $\lim_{\delta x \to 0} \frac{H(x+\delta x) - H(x)}{H_{TLM}(x, \delta x)} = 1$

Where, H_{TLM} is δy from $\nabla H(x) \cdot \delta x$

Thus,

```python
# H_TLM: The code that commputes delta_y given jacob of H evalulated at x, then @ delta_x
def H_TLM(x, delta_x):
    return H_jacobian(x).T @ delta_x
```

Then, can compute the expression for tangent test with:

```python
# Tangent Linear, the full numerator over denominator function
def tangent_test(x, delta_x):
    return (H(x + delta_x) - H(x)) / H_TLM(x, delta_x)
```

Adjoint: want difference between

$$LHS = H_{TLM}(x, \delta x)^T H_{TLM}(x, \delta x) \quad \text{and} \quad RHS = \delta x^T H_{Adj}(x, \delta y)$$

Where H_{adj} is computing $H(x)^T \cdot \delta y$ where $\delta y = H_{TLM}(x, \delta x)$

Thus, adjoint func:

```
# H_adj: The code that computes delta_x as H.T @ delta_y using H.T evaluated at x
def H_adjoint(x, delta_y):
    return H_jacobian(x) @ delta_y
```

So can do adjoint test error with:

```
# Adjoint, returns the error between LHS and RHS of the adjoint equation
# Where delta_y is obtained from H_TLM(x, delta_x)
def adjoint_test(x, delta_x):
    LHS = H_TLM(x, delta_x).T @ H_TLM(x, delta_x)
    delta_y = H_TLM(x, delta_x)
    RHS = delta_x.T @ H_adjoint(x, delta_y)
    return LHS - RHS
```

Evaluating both target_test and adjoint test w/

$x_0 = [0\ 0\ 1]$ and $\delta x = [0.01, 0.01, 0.01]$

Our program gives:

error b/w LHS and RHS

Forward: $y = 1.693$ Target test: 1.00998 Adjoint test: 0.0

yes, limit ≈ 1, and error $= 0$!

Now try $x_0 = [1, 1, 1]$:

```
x0: [[1. 1. 1.]]
Forwards Model: [3.41142901]
Tangent Test: [[1.01280912]]
Adjoint Test: [[0.]]
```

and $x_0 = [-1, -1, -1]$:

```
x0: [[-1. -1. -1.]]
Forwards Model: [3.41142901]
Tangent Test: [[0.98746727]]
Adjoint Test: [[0.]]
```

slight error from 1

In all cases the $\lim_{\delta x \to 0}$ from tangent test is ≈ 1 and the adjoint $= 0$. Perfect and satisfied tests.